7.5 Implementing RSA in Python

In the previous section we defined the RSA cryptosystem that used both a public key and private key to send encrypted messages between two parties. In this section, we will see how to implement the RSA cryptosystem in Python. First, we will see how to generate a private key when given two prime numbers. Second, we will see how to encrypt and decrypt a single number. Finally, we will see how to encrypt and decrypt text.

Key generation

Here is our implementation of the first phase of RSA: generating the public-private key pair. In this implementation, we will assume that the prime numbers p and q are given.[1]

[1] Algorithms do exist for generating these prime numbers, we just won't go over them here.

```
def rsa_generate_key(p: int, q: int) -> \
        tuple[tuple[int, int, int], tuple[int, int]]:
    """Return an RSA key pair generated using primes p and q.

    The return value is a tuple containing two tuples:
      1. The first tuple is the private key, containing (p, q, d).
      2. The second tuple is the public key, containing (n, e).

    Preconditions:
        - p and q are prime
        - p != q
    """
    # Compute the product of p and q
    n = p * q

    # Choose e such that gcd(e, phi_n) == 1.
    phi_n = (p - 1) * (q - 1)

    # Since e is chosen randomly, we repeat the random choice
    # until e is coprime to phi_n.
    e = random.randint(2, phi_n - 1)
    while math.gcd(e, phi_n) != 1:
        e = random.randint(2, phi_n - 1)

    # Choose d such that e * d % phi_n = 1.
    # Notice that we're using our modular_inverse from our work in the last
        chapter!
    d = modular_inverse(e, phi_n)

    return ((p, q, d), (n, e))
```

The algorithm makes use of both a `while` loop and the `random` module. The `random` module is used to generate an `e`, but the while loop ensures that the `e` we finally choose is valid. That is, we continue to randomly generate an `e` *until* `e` and `phi_n` have a greatest common divisor of 1. Once we have `e`, we can finally calculate the last part of our private key: `d`. To do so, we make use of the `modular_inverse` function we defined in the last chapter (which, in turn, used the `extended_gcd` function).

Encrypting and decrypting a number

Next, let's look at RSA encryption, which only uses the public key. Recall that the plaintext here is a number m between 1 and $n - 1$ inclusive, and the ciphertext is another number $c = m^e \% n$. This mathematical definition translates directly into Python:

```
def rsa_encrypt(public_key: tuple[int, int], plaintext: int) -> int:
    """Encrypt the given plaintext using the recipient's public key.

    Preconditions:
        - public_key is a valid RSA public key (n, e)
        - 0 < plaintext < public_key[0]
    """
    n, e = public_key

    encrypted = (plaintext ** e) % n

    return encrypted
```

The implementation for RSA decryption is almost identical, except we use the private key (i.e., `d`) for exponentiation.

```
def rsa_decrypt(private_key: tuple[int, int, int]  ciphertext: int) -> int:
    """Decrypt the given ciphertext using the recipient's private key.

    Preconditions:
        - private_key is a valid RSA private key (p, q, d)
        - 0 < ciphertext < private_key[0] * private_key[1]
    """
    p, q, d = private_key
    n = p * q

    decrypted = (ciphertext ** d) % n

    return decrypted
```

Encrypting and decrypting text using RSA

The above implementation of RSA is correct, but is a little unsatisfying because it encrypts numbers instead of strings, like we saw with the Caesar cipher and One-Time Pad cryptosystem. So next, we'll see how to adapt the RSA encryption and decryption algorithms to strings.

Our strategy will be to take a string and break it up into individual characters and encrypt each character, just as we did with the Caesar cipher. We'll use this approach for both encryption and decryption, using `ord`/`chr` to convert between characters and numbers, and a string accumulator to keep track of the encrypted/decrypted results.

```
def rsa_encrypt_text(public_key: tuple[int, int], plaintext: str) -> str:
    """Encrypt the given plaintext using the recipient's public key.

    Preconditions:
        - public_key is a valid RSA public key (n, e)
        - all({0 < ord(c) < public_key[0] for c in plaintext})
    """
    n, e = public_key

    encrypted = ''
    for letter in plaintext:
        # Note: we could have also used our rsa_encrypt function here instead
        encrypted = encrypted + chr((ord(letter) ** e) % n)

    return encrypted


def rsa_decrypt_text(private_key: tuple[int, int, int], ciphertext: str) ->
         str:
    """Decrypt the given ciphertext using the recipient's private key.

    Preconditions:
        - private_key is a valid RSA private key (p, q, d)
        - all({0 < ord(c) < private_key[0] * private_key[1] for c in
         ciphertext})
    """
    p, q, d = private_key
    n = p * q

    decrypted = ''
    for letter in ciphertext:
        # Note: we could have also used our rsa_decrypt function here instead
        decrypted = decrypted + chr((ord(letter) ** d) % n)

    return decrypted
```